CHAPTER 7

Writing Simple Shell Scripts

IN THIS CHAPTER

Working with shell scripts

Doing arithmetic in shell scripts

Running loops and cases in shell scripts

Creating simple shell scripts

You'd never get any work done if you typed every command that needs to be run on your Linux system when it starts. Likewise, you could work more efficiently if you grouped together sets of commands that you run all the time. Shell scripts can handle these tasks.

A *shell script* is a group of commands, functions, variables, or just about anything else you can use from a shell. These items are typed into a plain-text file. That file can then be run as a command. Linux systems have traditionally used system initialization shell scripts during system startup to run commands needed to get services going. You can create your own shell scripts to automate the tasks that you need to do regularly.

For decades, building shell scripts was the primary skill needed to join together sets of tasks in UNIX and Linux systems. As demands for configuring Linux systems grew beyond single-system setups to complex, automated cluster configurations, more structured methods have arisen. These methods include Ansible playbooks and Kubernetes YAML files, described later in cloud-related chapters. That said, writing shell scripts is still the best next step from running individual commands to building repeatable tasks in Linux systems.

This chapter provides a rudimentary overview of the inner workings of shell scripts and how they can be used. You learn how simple scripts can be harnessed to a scheduling facility (such as `cron` or `at`) to simplify administrative tasks or just run on demand as they are needed.

Understanding Shell Scripts

Have you ever had a task that you needed to do over and over that took lots of typing on the command line? Do you ever think to yourself, "Wow, I wish I could just type one command to do all this"? Maybe a shell script is what you're after.

Linux® Bible, Tenth Edition. Christopher Negus.

Shell scripts are the equivalent of batch files in Windows and can contain long lists of commands, complex flow control, arithmetic evaluations, user-defined variables, user-defined functions, and sophisticated condition testing. Shell scripts are capable of handling everything from simple one-line commands to something as complex as starting up a Linux system. Although dozens of different shells are available in Linux, the default shell for most Linux systems is called bash, the **B**ourne **A**gain **SH**ell.

Executing and debugging shell scripts

One of the primary advantages of shell scripts is that they can be opened in any text editor to see what they do. A big disadvantage is that large or complex shell scripts often execute more slowly than compiled programs. You can execute a shell script in two basic ways:

- The filename is used as an argument to the shell (as in `bash myscript`). In this method, the file does not need to be executable; it just contains a list of shell commands. The shell specified on the command line is used to interpret the commands in the script file. This is most common for quick, simple tasks.
- The shell script may also have the name of the interpreter placed in the first line of the script preceded by `#!` (as in `#!/bin/bash`) and have the execute bit of the file containing the script set (using `chmod +x filename`). You can then run your script just as you would any other program in your path simply by typing the name of the script on the command line.

When scripts are executed in either manner, options for the program may be specified on the command line. Anything following the name of the script is referred to as a *command-line argument*.

As with writing any software, there is no substitute for clear and thoughtful design and lots of comments. The pound sign (#) prefaces comments and can take up an entire line or exist on the same line after script code. It is best to implement more complex shell scripts in stages, making sure that the logic is sound at each step before continuing. Here are a few good, concise tips to make sure that things are working as expected during testing:

- In some cases, you can place an `echo` statement at the beginning of lines within the body of a loop and surround the command with quotes. That way, rather than executing the code, you can see what will be executed without making any permanent changes.
- To achieve the same goal, you can place dummy `echo` statements throughout the code. If these lines get printed, you know the correct logic branch is being taken.
- You can use `set -x` near the beginning of the script to display each command that is executed or launch your scripts using

```
$ bash -x myscript
```

- Because useful scripts have a tendency to grow over time, keeping your code readable as you go along is extremely important. Do what you can to keep the logic of your code clean and easy to follow.

Understanding shell variables

Often within a shell script, you want to reuse certain items of information. During the course of processing the shell script, the name or number representing this information may change. To store information used by a shell script in such a way that it can be easily reused, you can set *variables*. Variable names within shell scripts are case sensitive and can be defined in the following manner:

```
NAME=value
```

The first part of a variable is the variable name, and the second part is the value set for that name. Be sure that the `NAME` and *`value`* touch the equal sign, without any spaces. Variables can be assigned from constants, such as text, numbers, and underscores. This is useful for initializing values or saving lots of typing for long constants. The following examples show variables set to a string of characters (`CITY`) and a numeric value (`PI`):

```
CITY="Springfield"
PI=3.14159265
```

Variables can contain the output of a command or command sequence. You can accomplish this by preceding the command with a dollar sign and open parenthesis, following it with a closing parenthesis. For example, `MYDATE=$(date)` assigns the output from the `date` command to the `MYDATE` variable. Enclosing the command in back-ticks (`` ` ``) can have the same effect. In this case, the `date` command is run when the variable is set and not each time the variable is read.

Escaping Special Shell Characters

Keep in mind that characters such as the dollar sign ($), back-tick (`), asterisk (*), exclamation point (!), and others have special meaning to the shell, which you will see as you proceed through this chapter. On some occasions, you want the shell to use these characters' special meaning and other times you don't. For example, if you typed `echo $HOME`, the shell would think that you meant to display the name of your home directory (stored in the `$HOME` variable) to the screen (such as `/home/chris`) because a `$` indicates a variable name follows that character.

If you wanted literally to show `$HOME`, you would need to escape the `$`. Typing `echo '$HOME' or echo \$HOME` would literally show `$HOME` on the screen. So, if you want to have the shell interpret a single character literally, precede it with a backslash (\). To have a whole set of characters interpreted literally, surround those characters with single quotes (').

Using double quotes is a bit trickier. Surround a set of text with double quotes if you want all but a few characters used literally. For example, with text surrounded with double quotes, dollar signs ($), back-ticks (`), and exclamation points (!) are interpreted specially, but other characters (such as an asterisk) are not. Type these three lines to see the different output (shown on the right):

```
echo '$HOME * `date`'    $HOME * `date`
echo "$HOME * `date`"    /home/chris * Tue Jan 21 16:56:52 EDT 2020
echo $HOME * `date`      /home/chris file1 file2 Tue Jan 21 16:56:52 EDT 2020
```

Using variables is a great way to get information that can change from computer to computer or from day to day. The following example sets the output of the `uname -n` command to the `MACHINE` variable. Then I use parentheses to set `NUM_FILES` to the number of files in the current directory by piping (|) the output of the `ls` command to the word count command (`wc -l`):

```
MACHINE=`uname -n`
NUM_FILES=$(/bin/ls | wc -l)
```

Variables can also contain the value of other variables. This is useful when you have to preserve a value that will change so that you can use it later in the script. Here, `BALANCE` is set to the value of the `CurBalance` variable:

```
BALANCE="$CurBalance"
```

NOTE

When assigning variables, use only the variable name (for example, `BALANCE`). When you reference a variable, meaning that you want the *value* of the variable, precede it with a dollar sign (as in `$CurBalance`). The result of the latter is that you get the value of the variable, not the variable name itself.

Special shell positional parameters

There are special variables that the shell assigns for you. One set of commonly used variables is called *positional parameters* or *command-line arguments*, and it is referenced as $0, $1, $2, $3. . .$*n*. $0 is special, and it is assigned the name used to invoke your script; the others are assigned the values of the parameters passed on the command line in the order they appeared. For instance, let's say that you had a shell script named `myscript` which contained the following:

```
#!/bin/bash
# Script to echo out command-line arguments
echo "The first argument is $1, the second is $2."
echo "The command itself is called $0."
echo "There are $# parameters on your command line"
echo "Here are all the arguments: $@"
```

Assuming that the script is executable and located in a directory in your `$PATH`, the following shows what would happen if you ran that command with `foo` and `bar` as arguments:

```
$ chmod 755 /home/chris/bin/myscript
$ myscript foo bar
The first argument is foo, the second is bar.
The command itself is called /home/chris/bin/myscript.
There are 2 parameters on your command line
Here are all the arguments: foo bar
```

As you can see, the positional parameter `$0` is the full path or relative path to `myscript`, `$1` is `foo`, and `$2` is `bar`.

Another variable, `$#`, tells you how many parameters your script was given. In the example, `$#` would be 2. The `$@` variable holds all of the arguments entered at the command line. Another particularly useful special shell variable is `$?`, which receives the exit status of the last command executed. Typically, a value of zero means that the command exited successfully, and anything other than zero indicates an error of some kind. For a complete list of special shell variables, refer to the `bash` man page.

Reading in parameters

Using the `read` command, you can prompt the user for information and store that information to use later in your script. Here's an example of a script that uses the `read` command:

```
#!/bin/bash
read -p "Type in an adjective, noun and verb (past tense): " adj1 noun1 verb1
echo "He sighed and $verb1 to the elixir. Then he ate the $adj1 $noun1."
```

In this script, after the script prompts for an adjective, noun, and verb, the user is expected to enter words that are then assigned to the `adj1`, `noun1`, and `verb1` variables. Those three variables are then included in a silly sentence, which is displayed on the screen. If the script were called `sillyscript`, here's an example of how it might run:

```
$ chmod 755 /home/chris/bin/sillyscript
$ sillyscript
Type in an adjective, noun and verb (past tense): hairy football danced
He sighed and danced to the elixir. Then he ate the hairy football.
```

Parameter expansion in bash

As mentioned earlier, if you want the value of a variable, you precede it with a `$` (for example, `$CITY`). This is really just shorthand for the notation `${CITY}`; curly braces are used when the value of the parameter needs to be placed next to other text without a space. Bash has special rules that allow you to expand the value of a variable in different ways. Going into all of the rules is probably overkill for a quick introduction to shell scripts, but the following list presents some common constructs you're likely to see in bash scripts that you find on your Linux system.

`${var:-value}`: If variable is unset or empty, expand this to *value*.

`${var#pattern}`: Chop the shortest match for *pattern* from the front of *var*'s value.

`${var##pattern}`: Chop the longest match for *pattern* from the front of *var*'s value.

`${var%pattern}`: Chop the shortest match for *pattern* from the end of *var*'s value.

`${var%%pattern}`: Chop the longest match for *pattern* from the end of *var*'s value.

Try typing the following commands from a shell to test how parameter expansion works:

```
$ THIS="Example"
$ THIS=${THIS:-"Not Set"}
$ THAT=${THAT:-"Not Set"}
$ echo $THIS
Example
$ echo $THAT
Not Set
```

In the examples here, the `THIS` variable is initially set to the word `Example`. In the next two lines, the `THIS` and `THAT` variables are set to their current values or to `Not Set`, if they are not currently set. Notice that because I just set `THIS` to the string `Example`, when I echo the value of `THIS` it appears as `Example`. However, because `THAT` was not set, it appears as `Not Set`.

Note

For the rest of this section, I show how variables and commands may appear in a shell script. To try out any of those examples, however, you can simply type them into a shell, as shown in the previous example.

In the following example, `MYFILENAME` is set to `/home/digby/myfile.txt`. Next, the `FILE` variable is set to `myfile.txt` and `DIR` is set to `/home/digby`. In the `NAME` variable, the filename is cut down simply to `myfile`; then, in the `EXTENSION` variable, the file extension is set to `txt`. (To try these out, you can type them at a shell prompt as in the previous example and echo the value of each variable to see how it is set.) Type the code on the left. The material on the right side describes the action.

`MYFILENAME=/home/digby/myfile.txt`: Sets the value of `MYFILENAME`

`FILE=${MYFILENAME##*/}`: `FILE` becomes `myfile.txt`

`DIR=${MYFILENAME%/*}`: `DIR` becomes `/home/digby`

`NAME=${FILE%.*}`: `NAME` becomes `myfile`

`EXTENSION=${FILE##*.}`: `EXTENSION` becomes `txt`

Performing arithmetic in shell scripts

Bash uses *untyped variables*, meaning it normally treats variables as strings of text, but you can change them on the fly if you want it to.

Bash uses *untyped variables*, meaning that you are not required to specify whether a variable is text or numbers. It normally treats variables as strings of text, so unless you tell it otherwise with `declare`, your variables are just a bunch of letters to bash. However, when you start trying to do arithmetic with them, bash converts them to integers if it can. This makes it possible to do some fairly complex arithmetic in bash.

Integer arithmetic can be performed using the built-in `let` command or through the external `expr` or `bc` commands. After setting the variable `BIGNUM` value to `1024`, the

three commands that follow would all store the value 64 in the RESULT variable. The bc command is a calculator application that is available in most Linux distributions. The last command gets a random number between 0 and 10 and echoes the results back to you.

```
BIGNUM=1024
let RESULT=$BIGNUM/16
RESULT=`expr $BIGNUM / 16`
RESULT=`echo "$BIGNUM / 16" | bc`
let foo=$RANDOM; echo $foo
```

Another way to grow a variable incrementally is to use $(()) notation with ++I added to increment the value of I. Try typing the following:

```
$ I=0
$ echo "The value of I after increment is $((++I))"
The value of I after increment is 1

$ echo "The value of I before and after increment is $((I++)) and $I"
The value of I before and after increment is 1 and 2
```

Repeat either of those commands to continue to increment the value of $I.

Note

Although most elements of shell scripts are relatively freeform (where white space, such as spaces or tabs, is insignificant), both let and expr are particular about spacing. The let command insists on no spaces between each operand and the mathematical operator, whereas the syntax of the expr command requires white space between each operand and its operator. In contrast to those, bc isn't picky about spaces, but it can be trickier to use because it does floating-point arithmetic.

To see a complete list of the kinds of arithmetic that you can perform using the let command, type **help let** at the bash prompt.

Using programming constructs in shell scripts

One of the features that makes shell scripts so powerful is that their implementation of looping and conditional execution constructs is similar to those found in more complex scripting and programming languages. You can use several different types of loops, depending on your needs.

The "if. . .then" statements

The most commonly used programming construct is conditional execution, or the if statement. It is used to perform actions only under certain conditions. There are several variations of if statements for testing various types of conditions.

The first if...then example tests if VARIABLE is set to the number 1. If it is, then the echo command is used to say that it is set to 1. The fi statement then indicates that the if statement is complete, and processing can continue.

```
VARIABLE=1
if [ $VARIABLE -eq 1 ] ; then
echo "The variable is 1"
fi
```

Instead of using `-eq`, you can use the equal sign (=), as shown in the following example. The = works best for comparing string values, while `-eq` is often better for comparing numbers. Using the `else` statement, different words can be echoed if the criterion of the `if` statement isn't met (`$STRING = "Friday"`). Keep in mind that it's good practice to put strings in double quotes.

```
STRING="Friday"
if [ $STRING = "Friday" ] ; then
echo "WhooHoo.  Friday."
else
echo "Will Friday ever get here?"
fi
```

You can also reverse tests with an exclamation mark (!). In the following example, if `STRING` is not `Monday`, then "`At least it's not Monday`" is echoed.

```
STRING="FRIDAY"
if [ "$STRING" != "Monday" ] ; then
   echo "At least it's not Monday"
fi
```

In the following example, `elif` (which stands for "else if") is used to test for an additional condition (for example, whether `filename` is a file or a directory).

```
filename="$HOME"
if [ -f "$filename" ] ; then
   echo "$filename is a regular file"
elif [ -d "$filename" ] ; then
   echo "$filename is a directory"
else
   echo "I have no idea what $filename is"
fi
```

As you can see from the preceding examples, the condition you are testing is placed between square brackets `[ ]`. When a test expression is evaluated, it returns either a value of 0, meaning that it is true, or a 1, meaning that it is false. Notice that the `echo` lines are indented. The indentation is optional and done only to make the script more readable.

Table 7.1 lists the conditions that are testable and is quite a handy reference. (If you're in a hurry, you can type **`help test`** on the command line to get the same information.)

TABLE 7.1 Operators for Test Expressions

Operator	What Is Being Tested?
`-a` *file*	Does the file exist? (same as `-e`)
`-b` *file*	Is the file a block special device?
`-c` *file*	Is the file character special (for example, a character device)? Used to identify serial lines and terminal devices.
`-d` *file*	Is the file a directory?
`-e` *file*	Does the file exist? (same as `-a`)
`-f` *file*	Does the file exist, and is it a regular file (for example, not a directory, socket, pipe, link, or device file)?
`-g` *file*	Does the file have the set group id (SGID) bit set?
`-h` *file*	Is the file a symbolic link? (same as `-L`)
`-k` *file*	Does the file have the sticky bit set?
`-L` *file*	Is the file a symbolic link?
`-n` *string*	Is the length of the string greater than 0 bytes?
`-O` *file*	Do you own the file?
`-p` *file*	Is the file a named pipe?
`-r` *file*	Is the file readable by you?
`-s` *file*	Does the file exist, and is it larger than 0 bytes?
`-S` *file*	Does the file exist, and is it a socket?
`-t` *fd*	Is the file descriptor connected to a terminal?
`-u` *file*	Does the file have the set user id (SUID) bit set?
`-w` *file*	Is the file writable by you?
`-x` *file*	Is the file executable by you?
`-z` *string*	Is the length of the string 0 (zero) bytes?
expr1 `-a` *expr2*	Are both the first expression and the second expression true?
expr1 `-o` *expr2*	Is either of the two expressions true?
file1 `-nt` *file2*	Is the first file newer than the second file (using the modification time stamp)?
file1 `-ot` *file2*	Is the first file older than the second file (using the modification time stamp)?
file1 `-ef` *file2*	Are the two files associated by a link (a hard link or a symbolic link)?
var1 `=` *var2*	Is the first variable equal to the second variable?
var1 `-eq` *var2*	Is the first variable equal to the second variable?
var1 `-ge` *var2*	Is the first variable greater than or equal to the second variable?

Continues

TABLE 7.1 *(continued)*

Operator	What Is Being Tested?
var1 -gt *var2*	Is the first variable greater than the second variable?
var1 -le *var2*	Is the first variable less than or equal to the second variable?
var1 -lt *var2*	Is the first variable less than the second variable?
var1 != *var2*	Is the first variable not equal to the second variable?
var1 -ne *var2*	Is the first variable not equal to the second variable?

There is also a special shorthand method of performing tests that can be useful for simple *one-command actions*. In the following example, the two pipes (||) indicate that if the directory being tested for doesn't exist (`-d dirname`), then make the directory (`mkdir $dirname`):

```
# [ test ] || action
# Perform simple single command if test is false
dirname="/tmp/testdir"
[ -d "$dirname" ] || mkdir "$dirname"
```

Instead of pipes, you can use two ampersands to test if something is true. In the following example, a command is being tested to see if it includes at least three command-line arguments:

```
# [ test ] && {action}
# Perform simple single action if test is true
[ $# -ge 3 ] && echo "There are at least 3 command line arguments."
```

You can combine the `&&` and `||` operators to make a quick, one-line `if...then...else`. The following example tests that the directory represented by `$dirname` already exists. If it does, a message says the directory already exists. If it doesn't, the statement creates the directory:

```
# dirname=mydirectory
[ -e $dirname ] && echo $dirname already exists || mkdir $dirname
```

The case command

Another frequently used construct is the `case` command. Similar to a `switch` statement in programming languages, this can take the place of several nested `if` statements. The following is the general form of the `case` statement:

```
case "VAR" in
   Result1)
      { body };;
   Result2)
      { body };;
```

```
   *)
      { body } ;;
esac
```

Among other things, you can use the `case` command to help with your backups. The following `case` statement tests for the first three letters of the current day (`case 'date +%a' in`). Then, depending on the day, a particular backup directory (`BACKUP`) and tape drive (`TAPE`) are set.

```
# Our VAR doesn't have to be a variable,
# it can be the output of a command as well
# Perform action based on day of week
case `date +%a` in
   "Mon")
         BACKUP=/home/myproject/data0
         TAPE=/dev/rft0
# Note the use of the double semi-colon to end each option
         ;;
# Note the use of the "|" to mean "or"
   "Tue" | "Thu")
         BACKUP=/home/myproject/data1
         TAPE=/dev/rft1
         ;;
   "Wed" | "Fri")
         BACKUP=/home/myproject/data2
         TAPE=/dev/rft2
         ;;
# Don't do backups on the weekend.
   *)

BACKUP="none"
         TAPE=/dev/null
         ;;
esac
```

The asterisk (*) is used as a catchall, similar to the `default` keyword in the C programming language. In this example, if none of the other entries are matched on the way down the loop, the asterisk is matched and the value of `BACKUP` becomes `none`. Note the use of `esac`, or `case` spelled backwards, to end the `case` statement.

The "for. . .do" loop

Loops are used to perform actions over and over again until a condition is met or until all data has been processed. One of the most commonly used loops is the `for...do` loop. It iterates through a list of values, executing the body of the loop for each element in the list. The syntax and a few examples are presented here:

```
for VAR in LIST
do
    { body }
done
```

The `for` loop assigns the values in *`LIST`* to *`VAR`* one at a time. Then, for each value, the *`body`* in braces between `do` and `done` is executed. `VAR` can be any variable name, and `LIST` can be composed of pretty much any list of values or anything that generates a list.

```
for NUMBER in 0 1 2 3 4 5 6 7 8 9
do
   echo The number is $NUMBER
done

for FILE in `/bin/ls`
do
   echo $FILE
done
```

You can also write it this way, which is somewhat cleaner:

```
for NAME in John Paul Ringo George ; do
   echo $NAME is my favorite Beatle
done
```

Each element in the `LIST` is separated from the next by white space. This can cause trouble if you're not careful because some commands, such as `ls -l`, output multiple fields per line, each separated by white space. The string `done` ends the `for` statement.

If you're a die-hard C programmer, bash allows you to use C syntax to control your loops:

```
LIMIT=10
# Double parentheses, and no $ on LIMIT even though it's a variable!
for ((a=1; a <= LIMIT ; a++)) ; do
  echo  "$a"
done
```

The "while. . .do" and "until. . .do" loops

Two other possible looping constructs are the `while...do` loop and the `until...do` loop. The structure of each is presented here:

```
while condition      until condition
do                   do
   { body }            { body }
done                 done
```

The `while` statement executes while the condition is true. The `until` statement executes until the condition is true—in other words, while the condition is false.

Here is an example of a `while` loop that outputs the number 0123456789:

```
N=0
while [ $N -lt 10 ] ; do
   echo -n $N
   let N=$N+1
done
```

Another way to output the number 0123456789 is to use an `until` loop as follows:

```
N=0
until [ $N -eq 10 ] ; do
   echo -n $N
   let N=$N+1
done
```

Trying some useful text manipulation programs

Bash is great and has lots of built-in commands, but it usually needs some help to do anything really useful. Some of the most common useful programs you'll see used are `grep`, `cut`, `tr`, `awk`, and `sed`. As with all of the best UNIX tools, most of these programs are designed to work with standard input and standard output, so you can easily use them with pipes and shell scripts.

The general regular expression parser

The name *general regular expression print* (`grep`) sounds intimidating, but `grep` is just a way to find patterns in files or text. Think of it as a useful search tool. Gaining expertise with regular expressions is quite a challenge, but after you master it, you can accomplish many useful things with just the simplest forms.

For example, you can display a list of all regular user accounts by using `grep` to search for all lines that contain the text `/home` in the `/etc/passwd` file as follows:

```
$ grep /home /etc/passwd
```

Or you could find all environment variables that begin with `HO` using the following command:

```
$ env | grep ^HO
```

NOTE

The `^` in the preceding code is the actual caret character, `^`, not what you'll commonly see for a backspace, `^H`. Type `^`, `H`, and `O` (the uppercase letter) to see what items start with the uppercase characters HO.

To find a list of options to use with the `grep` command, type **`man grep`**.

Remove sections of lines of text (cut)

The `cut` command can extract fields from a line of text or from files. It is very useful for parsing system configuration files into easy-to-digest chunks. You can specify the field separator you want to use and the fields you want, or you can break up a line based on bytes.

The following example lists all home directories of users on your system. This `grep` command line pipes a list of regular users from the `/etc/passwd` file and displays the sixth field (`-f6`) as delimited by a colon (`-d':'`). The hyphen at the end tells `cut` to read from standard input (from the pipe).

```
$ grep /home /etc/passwd | cut  -d':' -f6 -
/home/chris
/home/joe
```

Translate or delete characters (tr)

The `tr` command is a character-based translator that can be used to replace one character or set of characters with another or to remove a character from a line of text.

The following example translates all uppercase letters to lowercase letters and displays the words `mixed upper and lower case` as a result:

```
$ FOO="Mixed UPpEr aNd LoWeR cAsE"
$ echo $FOO | tr [A-Z] [a-z]
mixed upper and lower case
```

In the next example, the `tr` command is used on a list of filenames to rename any files in that list so that any tabs or spaces (as indicated by the `[:blank:]` option) contained in a filename are translated into underscores. Try running the following code in a test directory:

```
for file in * ; do
   f=`echo $file | tr [:blank:] [_]`
   [ "$file" = "$f" ] || mv -i -- "$file" "$f"
done
```

The stream editor (sed)

The `sed` command is a simple scriptable editor, so it can perform only simple edits, such as removing lines that have text matching a certain pattern, replacing one pattern of characters with another, and so on. To get a better idea of how `sed` scripts work, there's no substitute for the online documentation, but here are some examples of common uses.

You can use the `sed` command essentially to do what I did earlier with the `grep` example: search the `/etc/passwd` file for the word `home`. Here the `sed` command searches the entire `/etc/passwd` file, searches for the word `home`, and prints any line containing the word `home`:

```
$ sed -n '/home/p' /etc/passwd
chris:x:1000:1000:Chris Negus:/home/chris:/bin/bash
joe:x:1001:1001:Joe Smith:/home/joe:/bin/bash
```

In this next example, `sed` searches the file `somefile.txt` and replaces every instance of the string `Mac` with `Linux`. Notice that the letter `g` is needed at the end of the substitution command to cause every occurrence of `Mac` on each line to be changed to `Linux`. (Otherwise, only the first instance of `Mac` on each line is changed.) The output is then sent to the `fixed_file.txt` file. The output from `sed` goes to `stdout`, so this command redirects the output to a file for safekeeping.

```
$ sed 's/Mac/Linux/g' somefile.txt > fixed_file.txt
```

You can get the same result using a pipe:

```
$ cat somefile.txt | sed 's/Mac/Linux/g' > fixed_file.txt
```

By searching for a pattern and replacing it with a null pattern, you delete the original pattern. This example searches the contents of the `somefile.txt` file and replaces extra blank spaces at the end of each line (`s/ *$`) with nothing (`//`). Results go to the `fixed_file.txt` file.

```
$ cat somefile.txt | sed 's/ *$//' > fixed_file.txt
```

Using simple shell scripts

Sometimes, the simplest of scripts can be the most useful. If you type the same sequence of commands repetitively, it makes sense to store those commands (once!) in a file. The following sections offer a couple of simple, but useful, shell scripts.

Telephone list

This idea has been handed down from generation to generation of old UNIX hacks. It's really quite simple, but it employs several of the concepts just introduced.

```
#!/bin/bash
# (@)/ph
# A very simple telephone list
# Type "ph new name number" to add to the list, or
# just type "ph name" to get a phone number

PHONELIST=~/.phonelist.txt

# If no command line parameters ($#), there
# is a problem, so ask what they're talking about.
if [ $# -lt 1 ] ; then
  echo "Whose phone number did you want? "
   exit 1
fi

# Did you want to add a new phone number?
if [ $1 = "new" ] ; then
  shift
  echo $* >> $PHONELIST
  echo $* added to database
  exit 0
fi

# Nope. But does the file have anything in it yet?
# This might be our first time using it, after all.
if [ ! -s $PHONELIST ] ; then
  echo "No names in the phone list yet! "
  exit 1
```

```
else
  grep -i -q "$*" $PHONELIST     # Quietly search the file
  if [ $? -ne 0 ] ; then         # Did we find anything?
    echo "Sorry, that name was not found in the phone list"
    exit 1
  else
    grep -i "$*" $PHONELIST
  fi
fi
exit 0
```

So, if you created the telephone list file as `ph` in your current directory, you could type the following from the shell to try out your `ph` script:

```
$ chmod 755 ph
$ ./ph new "Mary Jones" 608-555-1212
Mary Jones 608-555-1212 added to database
$ ./ph Mary
Mary Jones 608-555-1212
```

The `chmod` command makes the `ph` script executable. The `./ph` command runs the `ph` command from the current directory with the `new` option. This adds Mary Jones as the name and 608-555-1212 as the phone number to the database (`$HOME/.phonelist.txt`). The next `ph` command searches the database for the name Mary and displays the phone entry for Mary. If the script works, add it to a directory in your path (such as `$HOME/bin`).

Backup script

Because nothing works forever and mistakes happen, backups are just a fact of life when dealing with computer data. This simple script backs up all of the data in the home directories of all of the users on your Fedora or RHEL system.

```
#!/bin/bash
# (@)/my_backup
# A very simple backup script
#

# Change the TAPE device to match your system.
# Check /var/log/messages to determine your tape device.

TAPE=/dev/rft0

# Rewind the tape device $TAPE
mt $TAPE rew
# Get a list of home directories
HOMES=`grep /home /etc/passwd | cut -f6 -d':'`
# Back up the data in those directories
tar cvf $TAPE $HOMES
# Rewind and eject the tape.
mt $TAPE rewoffl
```

Summary

Writing shell scripts gives you the opportunity to automate many of your most common system administration tasks. This chapter covered common commands and functions that you can use in scripting with the bash shell. It also provided some concrete examples of scripts for doing backups and other procedures.

In the next chapter, you transition from learning about user features into examining system administration topics. Chapter 8, "Learning System Administration," covers how to become the root user, as well as how to use administrative commands, monitor log files, and work with configuration files.

Exercises

Use these exercises to test your knowledge of writing simple shell scripts. These tasks assume you are running a Fedora or Red Hat Enterprise Linux system (although some tasks work on other Linux systems as well). If you are stuck, solutions to the tasks are shown in Appendix B (although in Linux, there are often multiple ways to complete a task).

1. Create a script in your `$HOME/bin` directory called `myownscript`. When the script runs, it should output information that appears as follows:

   ```
   Today is Sat Jan 4 15:45:04 EST 2020.
   You are in /home/joe and your host is abc.example.com.
   ```

   Of course, you need to read in your current date/time, current working directory, and hostname. Also, include comments about what the script does and indicate that the script should run with the `/bin/bash` shell.

2. Create a script that reads in three positional parameters from the command line, assigns those parameters to variables named `ONE`, `TWO`, and `THREE`, respectively, and outputs that information in the following format:

   ```
   There are X parameters that include Y.
   The first is A, the second is B, the third is C.
   ```

   Replace X with the number of parameters and Y with all parameters entered. Then replace A with the contents of variable `ONE`, B with variable `TWO`, and C with variable `THREE`.

3. Create a script that prompts users for the name of the street and town where they grew up. Assign town and street to variables called `mytown` and `mystreet`, and output them with a sentence that reads as shown below (of course, `$mystreet` and `$mytown` will appear with the actual town and street the user enters):

   ```
   The street I grew up on was $mystreet and the town was
   $mytown
   ```

4. Create a script called `myos` that asks the user, "What is your favorite operating system?" Output an insulting sentence if the user types "Windows" or "Mac." Respond "Great choice!" if the user types "Linux." For anything else, say "Is <*what is typed in*> an operating system?"
5. Create a script that runs through the words *moose*, *cow*, *goose*, and *sow* through a `for` loop. Have each of those words appended to the end of the line "I have a. . . ."